April 2019

Investigating the Tradeoffs of GPUs for Parallel Processes

Jacob Freise
Worcester Polytechnic Institute

Investigating the Tradeoffs of GPUs for Parallel Processes

A Major Qualifying Project

Submitted to the Faculty of

Worcester Polytechnic Institute in partial

fulfillment of the requirements for the Degree

in Bachelor of Science in

Computer Science

By

_____

Jacob Freise

Date: 3/1/19

Sponsoring Organization:

Worcester Polytechnic Institute

Project Advisors:

_____

Professor Tian Guo, Advisor

Abstract

The goal of this project is to investigate the performance tradeoffs between a Central Processing Unit (CPU) when compared to general-purpose computing on GPUs (GPGPU). GPUs typically handle computer graphics, GPGPU is when a GPU is used to perform applications traditionally handled by a CPU. A comparison is achieved by running similar algorithms with varying input sizes on both pieces of hardware and measuring the differences in total execution time. The best suited applications for these types of tests are applications that benefit from parallel execution, to take advantage of the parallel nature of GPUs.

Acknowledgments

Without help from certain individuals the completion of this project would not have been possible. Some of these people helped us by providing guidance while others helped by supplying the resources necessary to proper testing.

I would like to thank Google for providing credits to use the Google Compute Engine. Without this funding testing would have been limited to a single low power GPU. This would have limited the results and made and findings inconclusive. With access the Google's Compute engine testing was allowed to take place on some of the most high end GPUs on the market today.

I would also like thank Tian Guo, the advisor to this project, for overseeing the project, providing much needed guidance, and giving me a sense of direction.

Table of Contents

List of Figures

Chapter 1: Introduction

GPUs have become a popular piece of hardware sold with nearly every high-end computer. They act as a coprocessor along with the standard CPU. However, much of the time they simple run idle. Perhaps, they can be utilized for processes outside of the standard use cases. The goal of this project is to understand GPU performance when used in general computing. This will help us better understand cloud-based GPUs as most of the tests will be done using a GPU provided through the use of the Google Compute Engine, an Infrastructure as a Service hosted by Google.

1.1 Motivation

Many applications require powerful Graphical Processing Units (GPU) to run well; CAD, graphics rendering, and machine learning all benefit from a better GPU. The problem is that GPUs are not only physically large but also expensive. This limits users to expensive desktop workstations to complete jobs that require such hardware. When the owner of the workstation is not making use of the GPU it simply sits there idle, creating an opportunity cost. Virtualization can help solve these issues by allowing users to access the computing power of a GPU from a remote location whenever they need it, and the idle time will be minimized since multiple users can access it. Nvidia Grid has implemented this system as GaaS, or Gaming as a service, where clients can have their video games rendered by a remote Graphics Card owned by Nvidia for a monthly fee. This way the user does not need to buy an expensive card to play a video game. CPU virtualization has been extensively researched, so to better understand how GPU virtualization can be efficiently implemented we must first look at the differences between a CPU and a GPU, which have fundamentally different design philosophies.

1.2 Report Organization

This report is broken into six distinct chapters. Chapter 1: Introduction, introduces the intent and purpose of the report, providing the motivation for the development of the project and a high-level overview of the differences between a CPU and GPU. Chapter 2: Literature review investigates previous work done on this topic, discussing their goal, methods and conclusions. Chapter 3: Proposed Design and Project Logistics discusses how the performance differences were going to be tested on the processing units. Chapter 4: Implementation discusses how the tests were implemented on the different processing units, and how the results were recorded. Chapter 5: Results discusses the results returned by the tests and what the tests signify. Chapter 6: Conclusions and future work summarizes the overall finding of this project and the next steps that can be taken to further the research in the differences between GPUs and CPUs.

Chapter 2: Background

CPUs originally were designed to have a single core with an extremely clock speed to ensure the fastest execution of a sequential program. This model was used for decades with each subsequent year resulting in faster single core performance. This meant that any programs would simply run faster with new hardware. However, since 2003 the energy consumption and heat dissipation caused by marginal increases in clock speed has imparted a hard performance limit. Microprocessor vendors switched to a multicore trajectory, seeking to maintain high execution speeds of sequential programs when moving onto multiple cores. This means that programs will not as easily see a speed up when the hardware is upgraded, it instead forces the software developer to develop with multicore processors in mind. CPUs make use of sophisticated control

logic to allow instructions from a single thread to execute in parallel or even out of sequential order while maintaining the appearance of sequential execution. They feature large cache memories to reduce the instruction and data access latencies of large complex applications.

GPUs are designed with a many-thread trajectory, focusing on the execution throughput of parallel applications. GPUs favor throughput, they operate at 10x the memory bandwidth of comparable CPUs. They move large amount of data in and out the main system memory. Reducing latency is much more expensive than increasing throughput in terms of power and chip area. The solution is to optimize for the execution of a massive number of threads. Small caches are used so multiple threads that access the same memory do not need to reassess the same DRAM. The sheer number of threads makes up for the potentially long execution time. This is called throughput-oriented design.

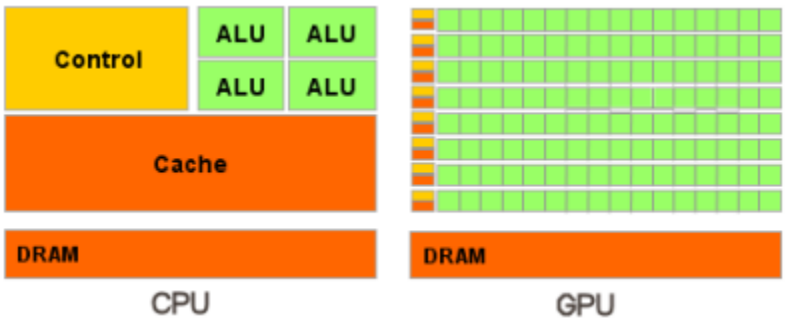

Figure 1: High level overview of the architectural designs of a CPU and a GPU demonstrating the differences in the layouts of the ALUs, Control Units, and Caches. [1]

The GPU has many more arithmetic logic units divided into rows, each with a small control unis and cache but the total cache size is much smaller than the CPU's.

Since hardware has reached a limit in terms of clock speed the only way for programs to enjoy a speed increase in future hardware generations is to make them more parallel. When an application is suitable for parallel execution an efficient implementation on a GPU can achieve a speedup factor of 100 over the speed of a single CPU core. The true speed up depends on how much the program can be parallelized. If 30% of a program can be parallelized, a 100x speed up of that portion will reduce the execution time by no more than 30%, making the entire application only about 1.4X faster. Even an infinite speed up of that portion take 30% off the the entire execution time, achieving a 1.43X speed up. The idea that speed up is limited by the proportion of the code than can be parallelized is called Amdahl's law. In practice, straightforward parallelization can cause a saturation of memory bandwidth, lowering the true speed up to only about 10x.

Most of the code in real applications tend to be sequential, making them better suited for CPU execution. However, only a small amount of execution time is spent in these sections, the rest is spent on parts the can be parallelized.

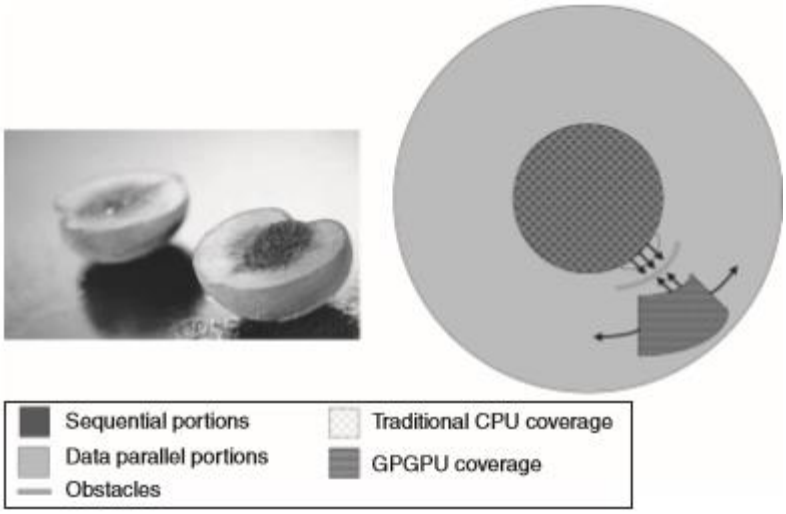

Figure 2: The "Peach Analogy" of modern algorithms the pit of the peach represents the execution time spent in serial processes. These serial processes are surrounded by the meat of the peach, representing a proportionally larger amount of execution time spent in parallelizable processes. [2]

An analogy is that of a peach, with the sequential execution time as a the pit of the peach and the parallelizable portions as the meat. The obstacles represent the problems faced by data transfer speeds from the main memory.

The purpose of parallelizing code is to make the end result run faster, but this can be a challenge because designing parallel code not only adds programming complexity but some parallel algorithms add large overheads that cause them to run slower than a similar sequential algorithm. The goal of this MQP is to determine which tasks are better suited for GPU execution when the overhead is considered.

2.1 Literature Review

As CPU performance increases begin to plateau due to the powerwall faced by designers trying to increase the clock speed and GPUs becoming more powerful every year, the question of

using a GPU to accelerate programs is becoming a more important one. Research on the topic is not exactly scarce, but it tends to get quickly outdated since GPU technology is growing quite rapidly.

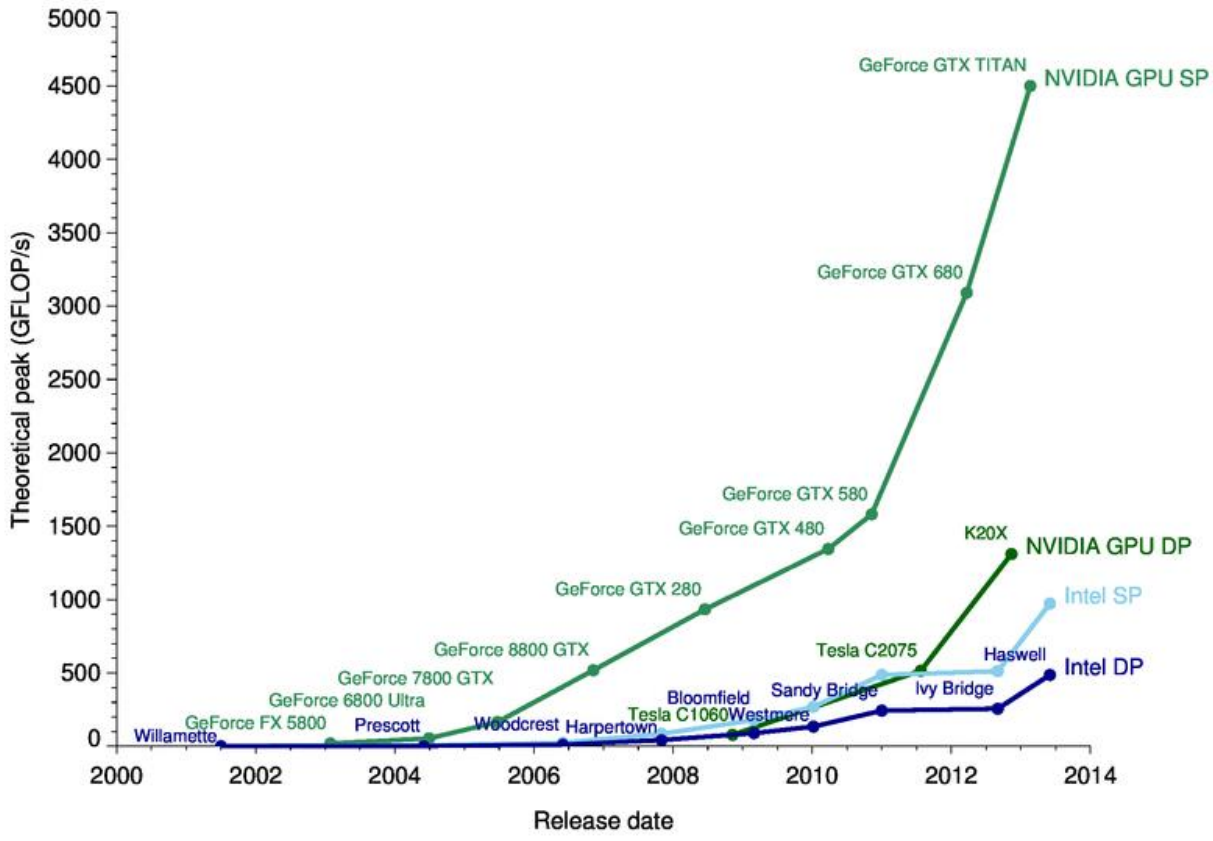

Figure 3: The theoretical peak performance of GPUs is following an exponential growth trend where CPU follows a more linear growth trend. [3]

GPU performance when compared to CPUs is not a new subject of research, partially because of the importance of the implications. Faster parallel programs allow for larger simulations, faster machine learning, and better graphics to name a few use cases. In order to find which types of benchmarks that should be ran in order to return the most insightful results some prior research on the subject was investigated.

It has been said before that running code on a GPU can offer up to a 1000X performance increase compared to running similar code on a CPU. The paper *Debunking the 100X GPU vs.*

CPU Myth: An Evaluation of Throughput Computing on CPU and GPU [4] attempts to investigate the validity of that claim. To test this the researchers developed some benchmarks that made high use of data parallelism, such as Monte Carlo, Fast Fourier Transform, and Basic Linear Algebra such as vector addition and multiplication. Using Intel's i7-960 and Nvidia's GTX 280, the researchers find that the performance differences between CPUs and GPUs are not as wide as initially thought. They found when proper optimizations are applied to both the CPU and GPU code they perform at roughly similar speeds. Instead of being 1000X faster, GPUs tend to operate closer to 2.5X faster when given highly parallelizable problems. These tests are using an old GPU relative to today, so perhaps the performance gap has grown.

Researchers have made it clear that optimization played a big part in how much speed up can actually be achieved using the GPU over the CPU. Code optimized for multithreaded CPU use can beat unoptimized GPU code and vice-versa. In the paper *Optimization schemes and performance evaluation of Smith–Waterman algorithm on CPU, GPU and FPGA* [5] researchers tried to measure the differences between speedup of optimized and unoptimized code on GPUs versus CPUs. They found that the optimized GPU was much faster than the unoptimized GPU version, but only marginally better than the CPU in various parallel benchmarks. The FPGA ended up being the fastest and most efficient due to its low power requirements.

We know that image processing is a highly parallel processes, this is due to the fact that each pixel can be operated on independently of every other pixel. The pixels do no depend on each other's prior state to find out their own next state. This means that image processing algorithms are a great candidate for GPU execution. In the paper *Performance comparison of FPGA, GPU and CPU in image processing* [6] researchers compare GPUs to CPUs and FPGAs using simple and well-known problems in image processing, two-dimensional filters, stereo-

vision and k-means clustering. In the two-dimensional filters test they found that GPUs far exceeded both CPUs and FPGAs. The CPU only beat the FPGA when the input size was kept small. The others two tests had FPGAs exceeding both the GPU and CPU, while the GPU and CPU were neck and neck. They concluded that the GPU only outperforms FPGAs when each pixel needs to be operated on independently, such as the case of the two-dimensional filter. The CPU used in the testing was an Intel Core 2 Extreme QX6850, the GPU was Nvidia's GTX 280, and the FPGA was a Xilinx XC4VLX160. As mentioned before this GPU is rather outdated, however I doubt that a more modern GPU will change the trends of their results in reference to the FPGA, however it might make the gap between the CPU and GPU larger.

One research group at WPI did a similar MQP called *Computing Performance Benchmarks among CPU, GPU, and FPGA* [7], where they compared the performance differences of CPUs, GPUs and FPGAs using a series of benchmarks. They chose seven benchmark suites that represented common computation problems, such as sorting and compression. They performed these across two Intel Xeon 5650 CPUs, the Virtex-5 FPGA and NVIDIA's GeForce GTX460 and 9800 GTX+ GPUs. Their conclusion was rather ambiguous, claiming that more research needs to be done in in the future as technology improves. The GPUs they chose are considered rather outdated compared to today's standards. So perhaps this is the future where technology has sufficiently improved they were talking about.

Investigating many benchmarks lead to an ambiguous conclusion to the varying results but investigating a specific task will lead to a much more explicit conclusion. The ideal task is something that can be broken down into many different parts that all are worked on independently. A research group at UC Berkeley in the paper *SSLShader: Cheap SSL Acceleration with Commodity Processors* [8] attempted a GPU implementation of the RSA

encryption algorithm. They claim that they developed the fastest known implementation of the algorithm using GPU parallelization. They achieved this by breaking up multiple RSA ciphertext messages and splitting them into thousands of threads so in order to utilize all the GPU cores.

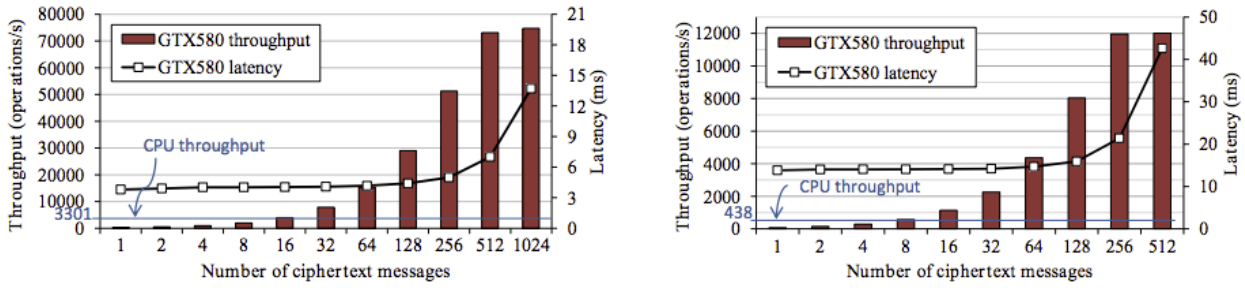

Figure 4: RSA MP performance on a GTX580. A single core (Xeon X5650 2.66 GHz) is used for CPU performance.

[4]

They found that the GPU execution outperformed the CPU execution as the number of messages increased. Their results show that the latency grows witch the throughput, this is due to the fact that as the input message increases, more data needs to be transcoded to the GPU. The larger bit cipher messages experience a faster growth in latency because they are taking up more bandwidth in the hardware. The GPU they chose was the Nvidia GTX 580, a brand-new card at the time. If this was ran on modern GPUs the trends would probably be similar, where the CPU outperforms the algorithm until the input size grows large enough. However, the rate of growth would probably be delayed or dampened due to modern GPUs having much more onboard memory.

A tree can be operated on independently, making them a good candidate for a parallel algorithm. *Efficient Parallel Graph Exploration on Multi-Core CPU and GPU* [9] was written for a conference on parallel architectures and investigated using a parallel approach for exploring breadth-first search trees. The team developed two algorithms, one that would run on a

conventional CPU, making sure to make use of all available cores, and an algorithm that runs on a GPU. The CPUs used were Intel's Xeon X5550, X5570, X7550, and E5345. The GPUs chosen were Nvidia's Fermi C2050, and Tesla GTX275. They found that a quad-core CPU operates very similarly in speed to that of a high-end GPU. However, the high-end GPUs used in their tests are considered quite weak compared to today's high-end GPUs, so perhaps the results would look different if ran on more contemporary GPUs.

In *Theano: A CPU and GPU math compiler in Python* [10] researchers attempt to create mathematical expression compiler in order to increase the speed at which they run. It forms low level machine code and supports CUDA kernels. They found that expressions running on GPUs tended to be faster compared to running on CPUs. They used a GTX 285 for their testing.

These research papers often seem to use a 200 series Nvidia card. According to Figure 3, this card has been highly outclassed by modern GPUs. This means that in any case where researchers found a CPU to be better suited to the problem could have a different result with newer hardware. This research has also pointed us in the direction of which benchmarks to use, especially highly parallel ones such as Linear Algebra and the Fast Fourier Transform.

Chapter 3: Proposed Design and Project Logistics

3.1 - Main Goal

The main goal of this project was to determine in which cases a GPU would be a better choice than a traditional CPU. A CPU excels in fast sequential operations, where GPU excels in massive throughput of parallel operations. Parallel operations are sequence independent, meaning that you do not need to solve the problem in a series of steps. You can paint the front of a house at the same time as you can paint the back of the house, making this a parallel problem, however, you cannot prime the house the same time as you paint it, making this a serial problem.

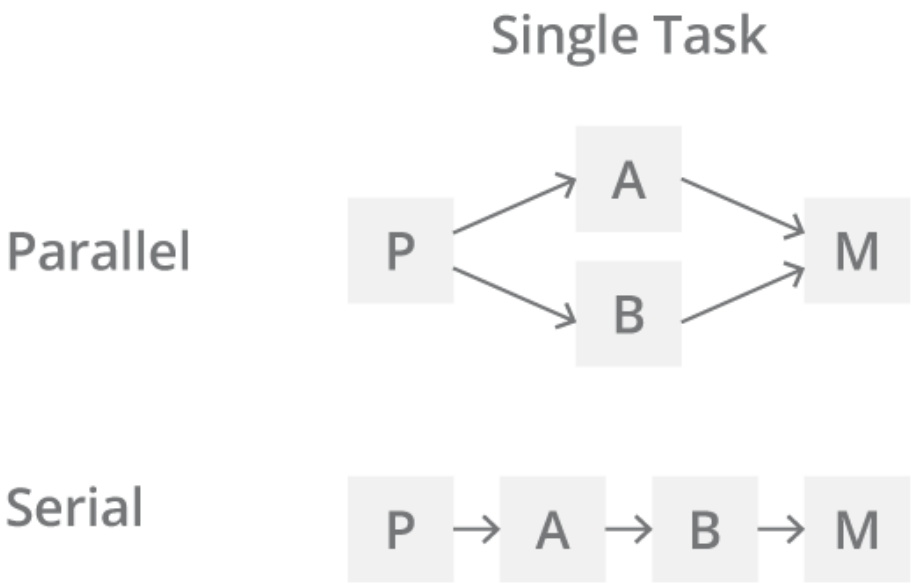

Figure 5: Example of the differences between a parallel and a serial process [11]

GPUs are better at handling parallel problems due to the large amount of independent processing units normally used in graphical problems, each pixel is independent of the ones around it. The CPU is better at handling sequential problems due to its fast clock speed and large memory cache per processing unit. This may make it seems that a GPU will always beat a CPU for a parallel task, however this is not the case. The primary reason being that data has to be transcoded from the main memory by the CPU to the GPU, this all takes some amount of time called overhead. If the total execution time on the GPU including overhead is larger than the total execution time of the same problem on the CPU, then using the GPU slowed down the total process time. The goal of this project is to develop a series of benchmarks that represent different use cases of parallel programs and provide each algorithm with varying sizes of data to determine when a GPU outperforms the CPU.

3.2 - Project Objectives

The main objective of the benchmarks was to get measurements of the CPU execution time, the GPU execution time, and the total overhead time required to transfer the data between them. In order to do this, the following needed to be achieved:

- Research and understand GPU design
- A way to create programs that can run on both a CPU and GPU
- A platform on which to test the programs on a variety of hardware
- Research which benchmarks should be chosen
- A system to capture the execution times and overhead costs for the CPUs and CPUs

3.3 - Project Management and Tasks

Since this was a solo project I was in charge of all research and development involved in the project. The first task was investigating how a GPU actually works, this involved reading several research papers and the book *Programming Massively Parallel Processors* by David B. Kirk and Wen-mei Hwu, engineers at Nvidia and Advanced Micro Devices respectively. This book provided much insight into the high-level architectural design of GPUs and its relevance to developing efficiently parallel programs. It also gave some simple examples I used for benchmarking purposes.

The next task was to research which platform to design the benchmarks should be chosen as there are a couple options available such as OpenCL and CUDA. The program needs to be ran on some hardware, the more hardware the stronger the results. My personal computer has a Nvidia 660M, this will be used as a preliminary test, but a more robust platform was needed. The choice of which programs be used to benchmark needed to be researched, as some benchmarks will return different results than others. Then the program would need to capture the amount of

time it took to run on all of the different pieces of hardware, and this data needed to be stored in a way that was able to show a clear trend if one existed.

3.4 - Design Decisions

Nvidia's CUDA was the programming model of choice. While CUDA is proprietary and is only supported by CUDA enabled GPUs limited to the Nvidia brand, it has the highest level of performance due to its tight hardware integration. The increased performance makes CUDA the primary choice for developers interested in parallel development, and as a result most fields interested in parallel development, such as machine learning, use CUDA. This makes the results of this report have the highest level of relevance. The main drawback of CUDA is that it is not supported on AMD GPUs.

The necessity of needing multiple pieces of hardware in order to get robust results was met through the use of Google's Compute Engine. Google's Compute Engine allows users to create servers using virtual CPUs and choose from a variety of very high-end GPUs. This allowed me to test any benchmarks on GPUs such as the K80, and P100, which would normally cost thousands of dollars.

The first benchmark chosen, vector addition, was out of simplicity, however it does represent a large portion of parallel problems. The benchmark simply adds two matrices of varying input sizes together. The second benchmark is a Fast Fourier Transform, this was chosen due to both the relevance and problem size. FFTs are used by a wide variety of industries and is a quite difficult problem that can be parallelized. This will allow us to show the difference between a simple problem and a much more difficult one.

The execution times are captured using CUDA's built in timer functions, even the CPU only code used these functions to keep the tests consistent as the execution time of timer functions could cause an effect on the measurements.

Chapter 4: Implementation

4.2 - Vector Addition

With the benchmarks chosen the next step is to implement each one into a form that could be tested by both the CPU and GPU of a system. CUDA allows for a program to be written primarily for a CPU but GPU kernels called for data to be offloaded to the GPU. With vector addition being the simplest benchmark that that was the obvious place to start. The textbook *Programming Massively Parallel Processes* [2] provided a simple example of vector addition which was expanded upon to allow for testing. The main features of the program generate two matrices of a given length, then run a simple while loop to add each of the respective indices together returning a third matrix. This represents a CPU executing vector addition. The system time was taken before and after the while loop using a CUDA timer function.

Creating the GPU version of the problem was a bit more involved. First several pointers needed to be created to hold the vectors in the memory of the GPU. Then data for our matrices was allocated on the GPU using the cudaMemcpy function. The cudaMemcpy allows for a data transfer from the CPU to GPU and vice versa. It takes four parameters; a pointer to the destination, a pointer to the source, the number of bytes being copied, and the direction of transfer. In this case the data was being transferred from the CPU to the GPU, so the destination were the pointers we had just created, and the source were the matrices that holding the data to be added. After copying the arrays to the GPU some more data was allocated for the matrix that

will hold the results of the addition. The function used for this was cudaMalloc, which functions similarly to the normal malloc function in standard C.

After preparing all of the necessary data onto the GPU the execution is performed by a GPU kernel, a segment of code that is run on every processing unit inside the GPU. The vector addition code looks like this:

*Int i = blockDim.x * blockIdx.x + threadIdx.x*

if(i<n) C[i] = A[i] + B[i];

CUDA kernels have access to two or more built-in variables, threadIDx and blockIDx that allows threads to distinguish themselves and determine the area of data each thread is to work on. Variable threadIdx gives each thread a unique coordinate within a block. In this case we are using a one-dimensional thread organization because out matrix is only one dimensional, therefore we only use threadIdx.x. All threads in a block share a common block coordinate. blockDim is a value picked by the developer which determines the amount of threads per block. If a developer chose a blockDim of 256 then blockIDx.x 0 would contain the threads 0-255. The value of n represents the number of values in the matrix. It prevents the GPU from attempting adding data that does not exist together by constraining it.[2]

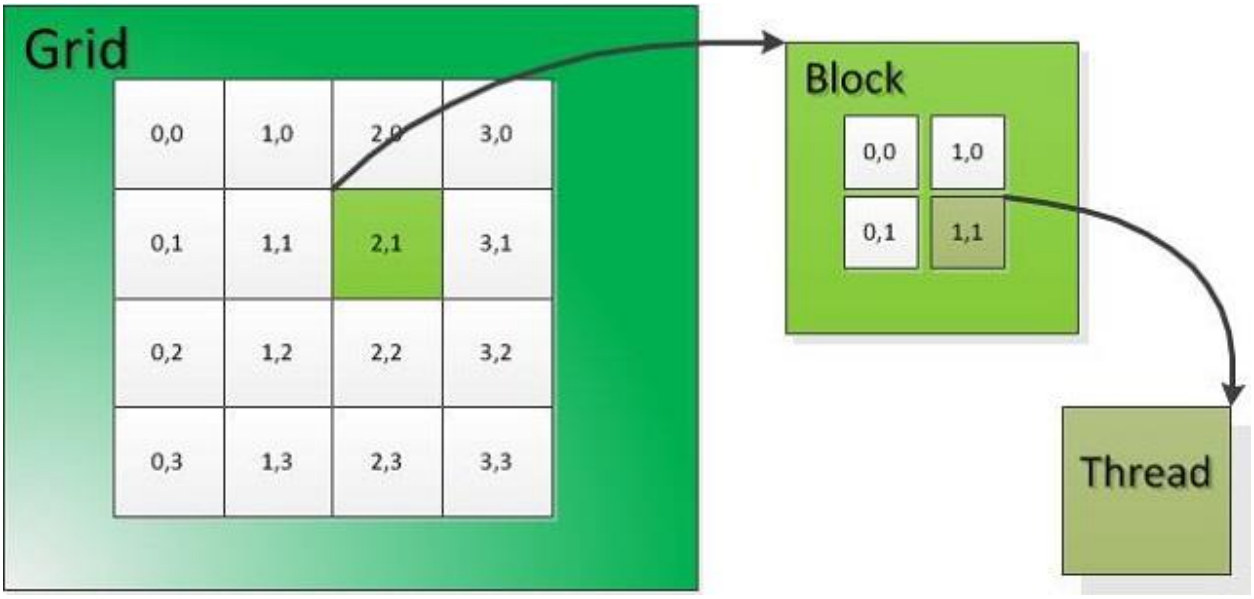

Figure 6: Example of a two dimensional threads and blocks. In this case the block dimension is 2, but the two dimensions allow each block to hold 4 threads. Not to be confused with the 4x4 grid size which is fixed by the hardware. [6]

4.3 - Fast Fourier Transform

The Fast Fourier Transform was a bit easier because I used a few common implementations and simply adjusted them to my needs. There is a well-known FFT program called FFTW3 which does simple FFTs on a CPU. It took a parameter representing the scale, the larger the scale the larger the computation time. Nvidia supplies an example project called cuFFT which took a similar scale parameter. A timer function captured the amount of time that each program took to calculate the FFT.

Chapter 5: Results

n	i7-3610QM	660M OH	660M NOH	vCPU	k80 OH	k80 NOH	vCPU2	P100 OH	P100 NOH
10	0.034	0.328	0.046	0.012	0.316	0.037	0.009	0.413	0.032
100	0.03	0.961	0.039	0.015	0.244	0.024	0.02	0.263	0.02
1000	0.028	0.963	0.043	0.015	0.242	0.025	0.01	0.273	0.034
10000	0.03	0.583	0.049	0.04	0.372	0.041	0.035	0.283	0.018
100000	0.03	2.094	0.426	0.42	0.722	0.034	0.314	0.456	0.022
1000000	0.061	7.616	1.551	3.661	2.916	0.129	3.423	2.309	0.042
10000000	0.073	54.628	12.752	38.65	22.652	1.012	35.593	19.353	0.252
100000000	0.077	504.04	110.795	396.33			372.82		

Figure 7: Total result chart of vector addition. OH stands for results including overhead times. NOH stands for results excluding overhead times. All results are times measured in (ms)

n	vCPU	k80 OH	k80 NOH	vCPU2	P100 OH	P100 NOH
10	0.001	453.995	0.037	0.001	361.812	0.051
100	0.001	424.52	0.032	0.001	301.719	0.024
1000	0.014	419.897	0.02	0.025	281.144	0.012
10000	0.136	406.131	0.02	0.193	309.633	0.035
100000	2.047	470.963	0.041	1.96	352.604	0.024

Figure 8: Total result chart of Fast Fourier transform. OH stands for results including overhead times. NOH stands for results excluding overhead times. All results are times measured in (ms)

Bytes	totalFFT OH	FFT NOH	vectorAdd OH	vectorAdd NOH	simpleVecadd NOH	simpleVecadd NOH2
128	331.77	0.039	0.319	0.033	0.346	0.312
848	292.494	0.038	0.274	0.024	0.267	0.236
32768	281.017	0.022	0.596	0.046	0.306	0.272
325632	294.9	0.022	4.053	0.071	0.574	0.536
3241984	343.611	0.041	39.664	0.234	2.45	2.337

Figure 9: Comparing different tests using the same amount of input data. vectorAdd is a custom vector addition program, and simpleVecadd is provided by Nvdidia. OH stands for results including overhead times. NOH stands for results excluding overhead times. All results are times measured in (ms)

5.1 - Vector Addition

The first test to determine the difference speed differences between the CPU and GPU was running a simple matrix addition test. Two matrices of n length were generated and then added together using both a CPU and then a GPU using a CUDA kernel. The experiment was measuring how long it took each length of matrix to complete its addition. The system time was captured before the vector addition and afterwards to calculate the total execution time. Execution time was the metric chosen because it has the most real-world implications. The faster the execution time the faster the program is in general. The time taken for the GPU to allocate and transfer data was also measured and included in the total execution time. This is an important metric because it shows how much of the total time used by the GPU was spent doing something that was not calculating. It can help demonstrate why the time taken by a GPU might be higher than expected. The CPU has no overhead time because it has direct access to the data. These were performed on three different graphics cards; a 660M, a K80, and a P100. The CPU was an i7-3610QM when paired with the 660M, but the K80 and P100 used an Intel Haswell virtual CPU provided by the Google Compute engine.

\

660M results

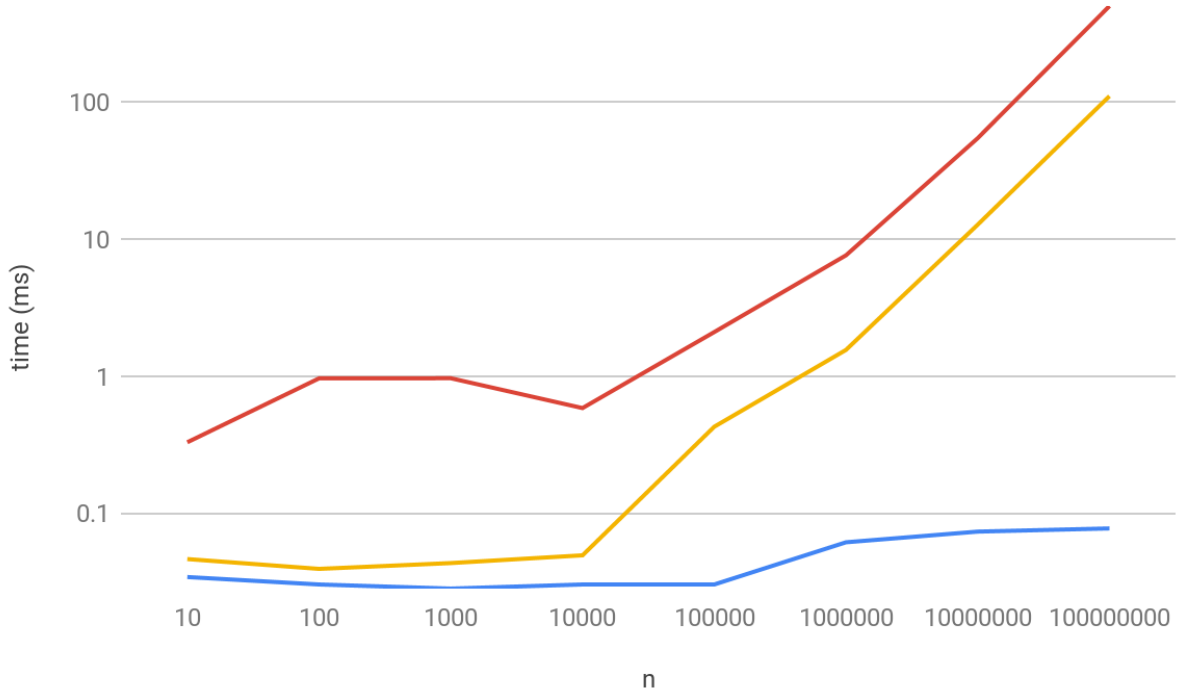

Figure 7: Results of a parallelized vector addition ran on an Nvidia 660M
Blue line represents the total CPU execution time
Yellow line represents the GPU execution time without measuring the overhead of data transfer
Red line represents the total GPU execution time with overhead

The CPU took relatively the same amount of time until n increased to over 1 million. This was probably due to n exceeding the size of the cache. The GPU execution followed a similar trend until 10,000 at which point it started growing linearly with n. This was also probably due to n exceeding the cache size. The total time with overhead followed the execution time trend, implying that the overhead did not grow as n increased. This is a case where the small cache size of the GPU negatively affects the performance relative the CPU.

K80 results

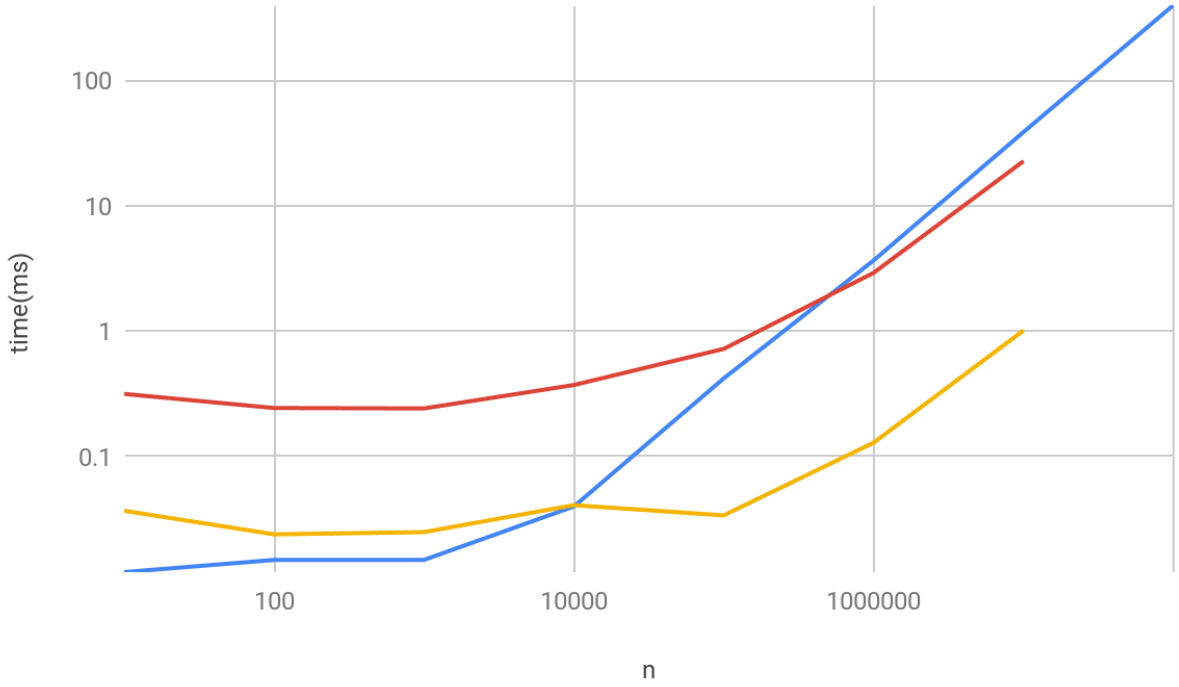

Figure 8: Results of a parallelized vector addition ran on a Nvidia K80
Blue line represents the total CPU execution time
Yellow line represents the GPU execution time without measuring the overhead of data transfer
Red line represents the total GPU execution time with overhead

Using a K80, a more powerful GPU, and a virtual CPU provided by Google's compute engine, the trends started to change. The CPU time started to increase linearly with n after n exceeded 10,000. This implies that the virtual CPU has a smaller cache than the i7-3610QM. The GPU execution time started to follow n in what looks like exponential growth but exceeding an n value of 1 million would cause the test to crash. The growth it probably follows a linear trend after fully exceeding the cache which looks like it happens when n reaches 100,000. The total time, including overhead, follows the execution time, implying that the overhead does not increase as n increases. This test shows that a higher quality GPU can change the trade-offs between a GPU and CPU performance differences. This is a case where the GPU offers better

performance over the CPU with a large enough data set, but a smaller data set still favors the CPU.

P100 results

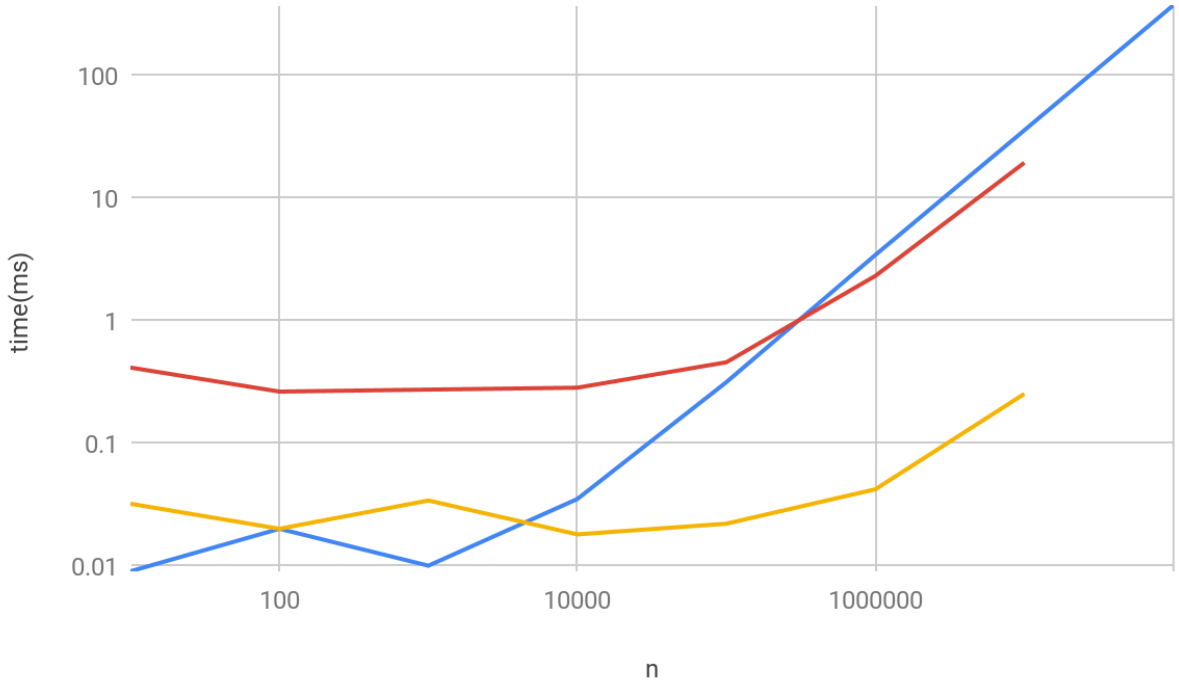

Figure 9: Results of a parallelized vector addition ran on an Nvidia P100
Blue line represents the total CPU execution time
Yellow line represents the GPU execution time without measuring the overhead of data transfer
Red line represents the total GPU execution time with overhead

Using an even more powerful GPU, the P100, and the same processor as the last test, the performance follows the same trends. The CPU grows linearly after exceeding the same n value, which is most likely due to n exceeding the cache size. The GPU follows the same trend as the K80's results, however the times are slightly faster. This implies that there are diminishing returns when it comes to a GPU, as after a certain point a more powerful one will not make a massive difference in total time.

These graphs help demonstrate the tradeoff between CPUs and GPUs. A CPU will handle a small data set more efficiently because the overhead of transferring it to the GPU is larger than the performance gain the GPU offers. One point of confusion created by these graphs is how the CPU results of the 660M are always faster than the GPU. This is probably due to the CPU having a cache larger than n.

5.2 - Fast Fourier Transform

The next test was comparing the performance between the GPU and CPUs was a Fast Fourier Transform. This test was set up similarly to the Vector Addition test where an increasing load was supplied to the algorithm and timers captured the execution times with and without the overhead used for data transfer. The n in this instance represented what is called the scale, a larger scale allows for more detailed results from the wave but increases computational complexity. Increasing the scale should increase the execution times in both the CPU and GPU. This was only performed on the K80 and P100 using Google's virtual CPU.

K80 results

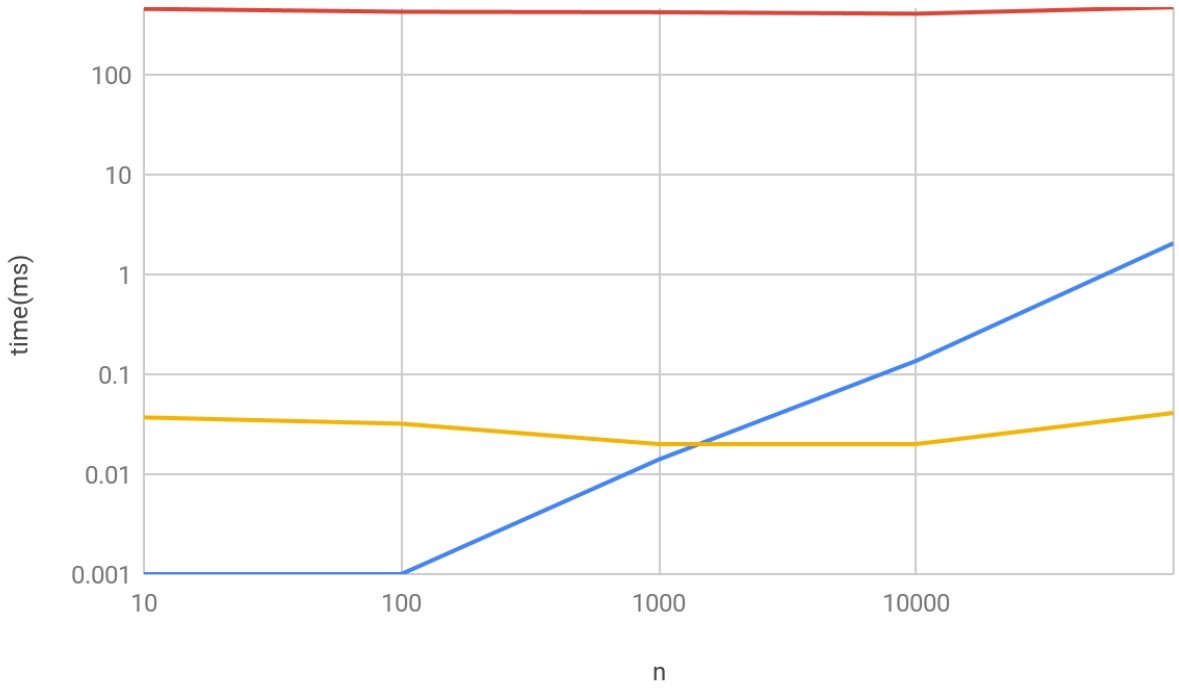

Figure 10: Results of a parallelized Fast Fourier Transform ran on a Nvidia K80
Blue line represents the total CPU execution time
Yellow line represents the GPU execution time without measuring the overhead of data transfer
Red line represents the total GPU execution time with overhead

The Fast Fourier transform on the CPU shows a linear growth after n equals 100, n represents the amount of data in this test, so it is probably where the cache size is exceeded. The GPU experiences no growth through the entire data set. This implies that a GPU will eventually beat a CPU given a large enough data set, however any attempts to increase the scale further resulted in a crash of the GPU program.

P100 results

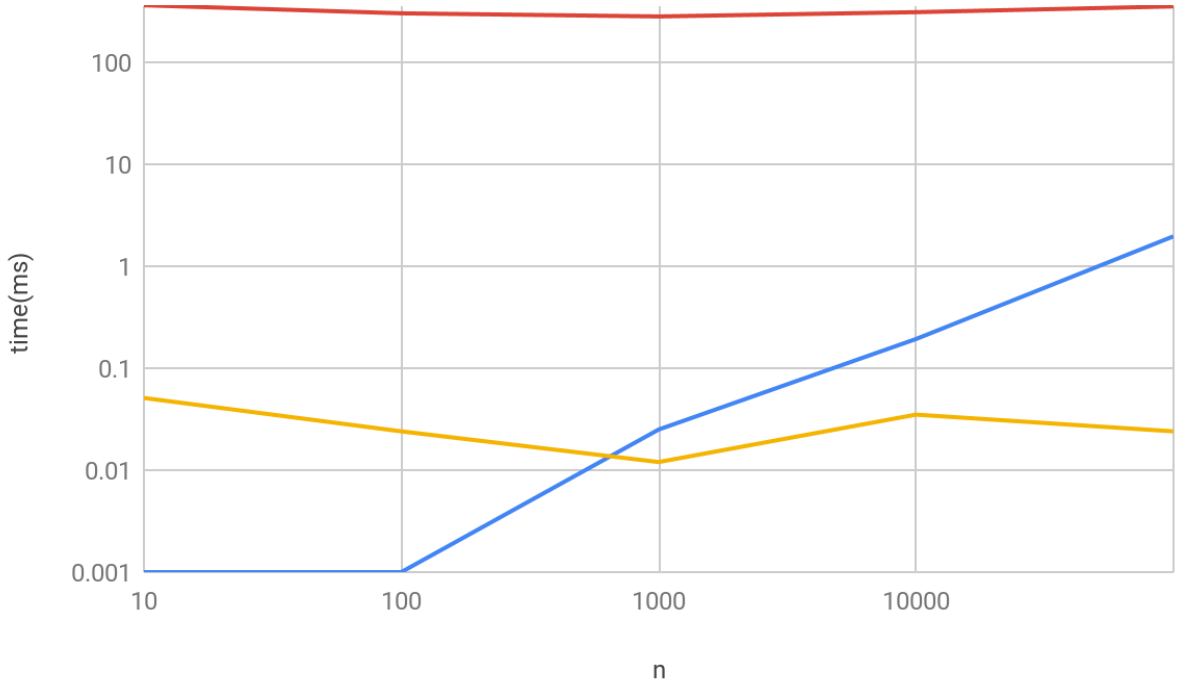

Figure 11: Results of a parallelized Fast Fourier Transform ran on a Nvidia P100
Blue line represents the total CPU execution time
Yellow line represents the GPU execution time without measuring the overhead of data transfer
Red line represents the total GPU execution time with overhead

These graphs were interesting because the GPU performance time does not increase as the sample size increases where the CPU experiences linear growth. This test again shows that the total execution time in the GPU does not increase as n grows. The results show that CPUs are faster, but according to the trends increasing the data set further would have the CPU take longer than the GPU. However, this could not be tested due to increasing n any further would cause the GPU test to crash.

5.3 - Result Comparison

The next test was comparing the performance difference between two tests given the same data size of the input. This was performed using only one GPU, the K80. The FFT testing error was fixed before this test.

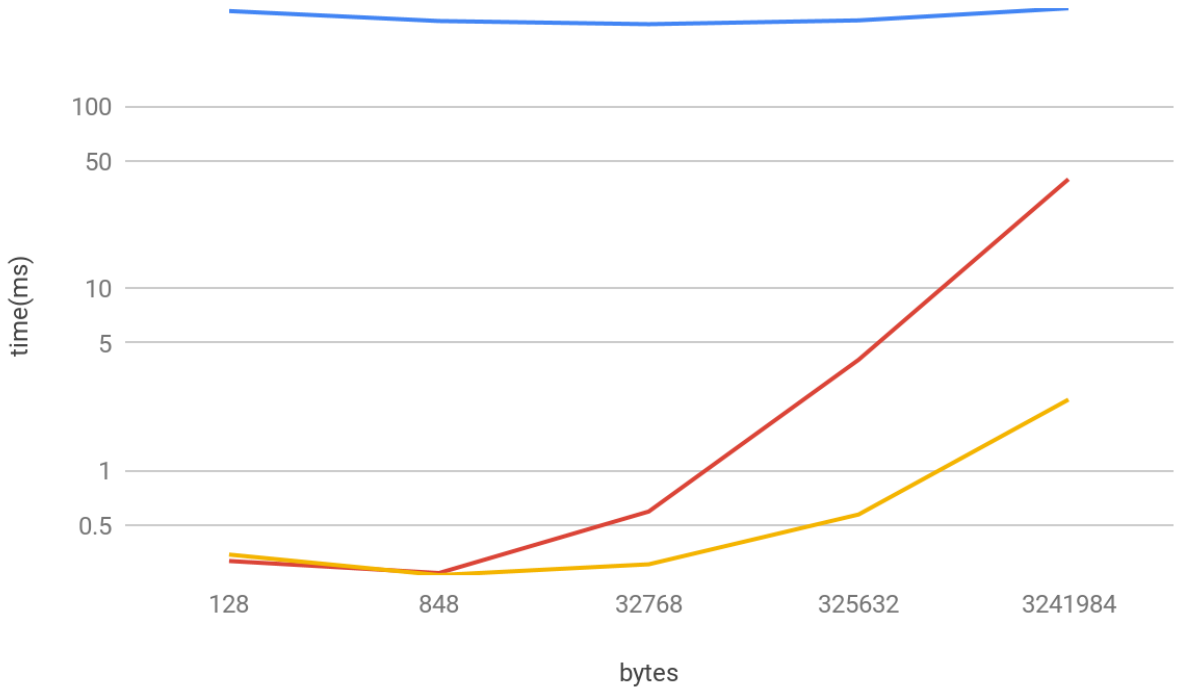

Figure 12: Comparisons of the total execution times of two vector addition algorithms and the FFT on a Nvidia K80
Blue line represents the total GPU FFT execution time with overhead
Yellow line represents the total GPU vector addition time with overhead with optimization
Red line represents the total GPU vector addition time with overhead without optimization

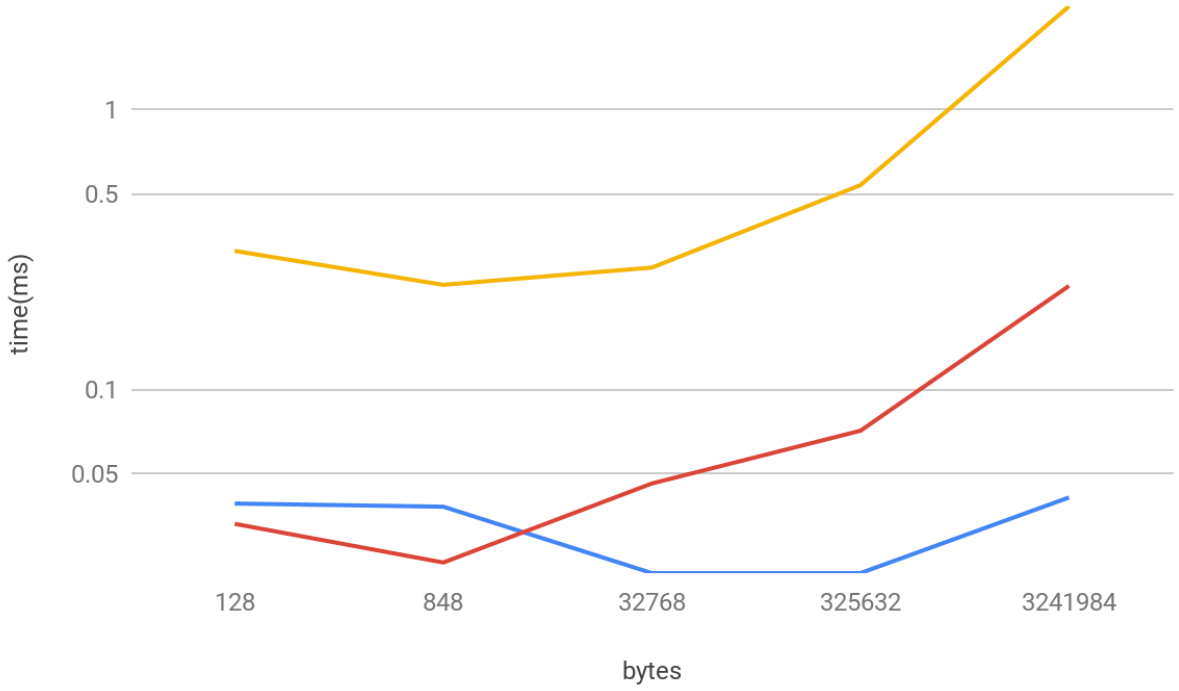

Figure 13: Comparisons of the execution times not including data transfer of the two vector addition algorithms and the FFT on an Nvidia K80
Blue line represents the total GPU FFT execution time with NO overhead(time spent in data transfer)
Yellow line represents the total GPU vector addition time with NO overhead with optimization
Red line represents the total GPU vector addition time with NO overhead without optimization

The graph shows how the execution times for the different tests were usually very similar, this speaks to the computation efficiency of the GPU. It also shows that the overhead of transferring the data is where this efficiency is lost. Notice how large the gap is between the FFT execution time and FFT total time. This graph does create some questions, why do certain processes take more overhead? Why does the overhead of vecAdd grow over time when simple Vecadd and FFT stay relatively the same?

5.4 - Data Analysis

In this experiment we try to understand the overheads of the workloads by using a command provided by Nvidia called Nvidia-SMI which shows the current state of the GPU.

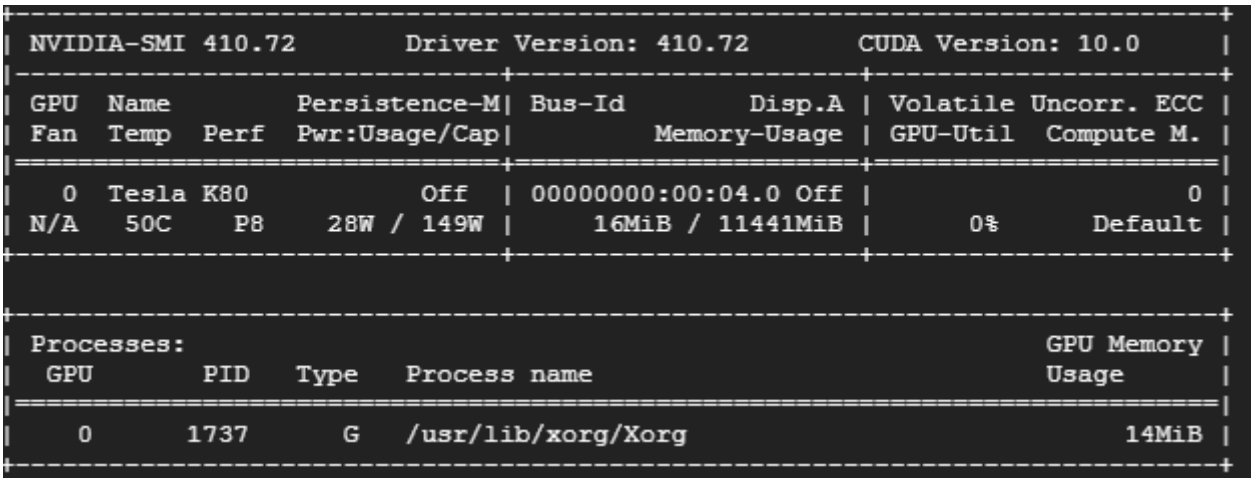

Figure 14: Output of the linux command Nvidia-SMI. This is used to get the current memory utilization of the GPU.

The relevant information is the total memory usage and the memory usage per process.

A script was written in order to run this test multiple times per second while a benchmark is being run. This way the memory usage can be graphed over time.

Using grep, each line containing the string MiB was saved to a text file along with the outputs from the benchmark script.

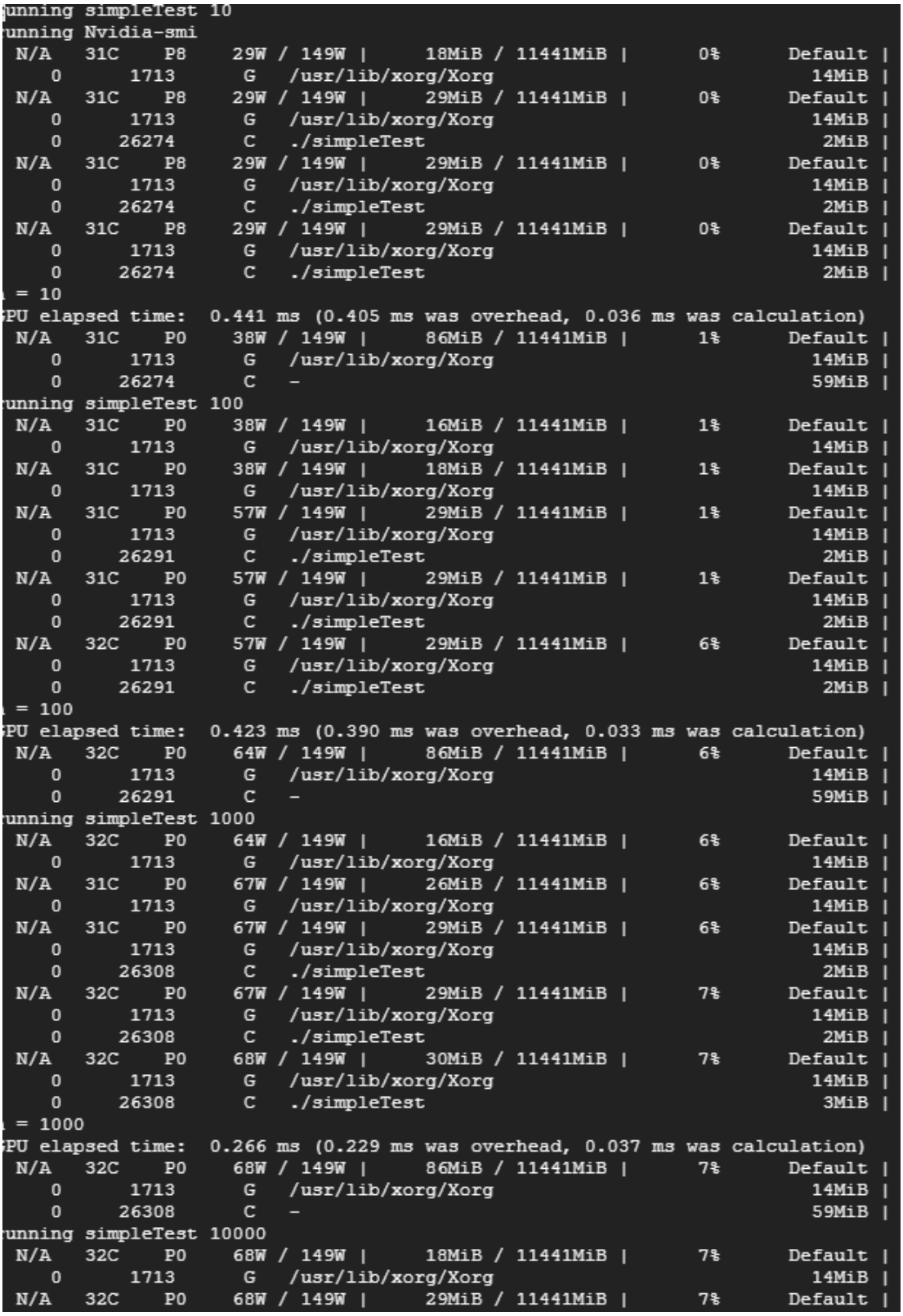

Figure 15: Output of running Nvidia-SMI during the Vector Addition algorithm and using grep to pull relevant data

This shows the utilization over time as the tests are being ran, as expected the larger the n value, the higher memory utilization.

Chapter 6: Conclusions and Future Work

According to the results it seems that the optimal processing unit depends on the size of the data set. In general, a small data set will take less time to complete when ran on a CPU because of the immediacy of the data. The CPU has a larger cache and a faster clock speed so if it can store the entire data set within the cache then it will be able to work through the data quite quickly. The GPU faces an overhead cost due to the time it takes to allocate and transfer the data to the GPU. A CPU can finish the execution before the GPU even receives the data if the data set is small. However, as the data set increases the overhead cost does not. This means that a large amount of data will take a similar amount of time to transfer to the GPU as a small set of data. Since CPUs experience a linear growth in execution time due to its sequential nature, there is a point where using the GPU becomes the faster option. This point is relative to the transfer speed of the GPU and the cache size of the CPU.

These tests were all operated on a discrete GPU, where the data must travel through PCIE to reach the GPU. A future work could perform similar benchmarks using an integrated GPU. This would be interesting as the integrated GPU is located inside the CPU, which would reduce the amount of overhead. Since overhead is the main limiting factor the tests would probably favor the GPU with smaller data sets. Recently the performance of integrated GPUs has increased making them competitive with discrete GPUs making them an interesting point of research for general purpose computing on graphical processing units.

Bibliography

[1] http://ixbtlabs.com/articles3/video/cuda-1-p1.html

[2] Kirk, David, and Wen-mei Hwu. *Programming Massively Parallel Processors: a Hands-on with CUDA*. Morgan Kaufmann Publishers, 2010.

[3] https://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html

[4] https://dl.acm.org/citation.cfm?id=1816021

[5]https://www.researchgate.net/publication/262312302_Optimization_schemes_and_performance_evaluation_of_Smith-Waterman_algorithm_on_CPU_GPU_and_FPGA

[6]https://www.researchgate.net/publication/220759541_Performance_comparison_of_FPGA_GPU_and_CPU_in_image_processing

[7]https://web.wpi.edu/Pubs/E-project/Available/E-project-030212-123508/unrestricted/Benchmarking_Final.pdf

[8] https://people.eecs.berkeley.edu/~sangjin/static/pub/nsdi2011_sslshader.pdf

[9] https://ppl.stanford.edu/papers/pact11-hong.pdf

[10] http://www.iro.umontreal.ca/~lisa/pointeurs/theano_scipy2010.pdf

[11] https://www.xait.com/resources/blog/serial-vs-parallel-process/

[12] https://sidkashyap.files.wordpress.com/2013/05/block_grid_thread.jpg

Appendix

VectorAddition.cu

```
#include <stdio.h>
#include <stdlib.h>

void cpuVectorAdd(int *a, int *b, int *c, int n)
{
    int i;

    for (i = 0; i < n; i++)
    {
        c[i] = a[i] + b[i];
    }
}

__global__ void gpuVectorAdd(int *a, int *b, int *c, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n)
    {
        c[i] = a[i] + b[i];
    }
}


void CPUtest(int *a, int *b, int*c, int n)
{
    float time;
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start, 0);

    cpuVectorAdd(a, b, c, n);
    cudaDeviceSynchronize();//simply there to balance the test

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
    printf("CPU elapsed time:  %3.3f ms \n", time);
```

```
}


void GPUtest(int *h_A, int *h_B, int*h_C, int n)
{
    int *d_A, *d_B, *d_C;
    int size = n * sizeof(int);

    int blockSize;
    int minGridSize;
    int gridSize;

    float time, sumTime, gpuTime, overheadTime;
    cudaEvent_t start, stop; //timer varaibles
    cudaEventCreate(&start); //initializes timer
    cudaEventCreate(&stop);

    cudaEventRecord(start, 0); //starts timer

    cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize, gpuVectorAdd, 0,
n); //calculates block size

    gridSize = (n + blockSize - 1) / blockSize; //calculates grid size

    cudaEventRecord(stop, 0); //stops timer
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop); //calculates elapsed time
    sumTime = time; //used to calculate total time

    //printf("Occupancy calculator elapsed time:  %3.3f ms \n", time);
    //printf("Grid size of: %d Block size of %d minGridSize: %d\n", gridSize,
blockSize, minGridSize);

    cudaEventRecord(start, 0); //start timer for allocation

    cudaMalloc((void **) &d_A, size); //allocates some space on GPU
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
    sumTime += time;
    //printf("cudaMalloc elapsed time:  %3.3f ms \n", time);
```

```
    cudaEventRecord(start, 0);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice); //moves  data from
cpumemory to GPU memory
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
    sumTime += time;
    //printf("cudaMemcpy Host to Device elapsed time:  %3.3f ms \n", time);


    cudaEventRecord(start, 0);

    gpuVectorAdd <<<gridSize, blockSize>>> (d_A, d_B, d_C, n); //calls kernel

    cudaDeviceSynchronize(); //makes sure kernel is done before moving on

    cudaEventRecord(stop, 0);

    cudaError_t error = cudaGetLastError(); //this is how i found out that i
needed to use special compile parameters
    if (error != cudaSuccess)
    {
        fprintf(stderr, "ERROR: %s\n", cudaGetErrorString(error));
        exit(-1);
    }

    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
    gpuTime = time;
    sumTime += time;
    //printf("vecAdd elapsed time:  %3.3f ms \n", time);

    cudaEventRecord(start, 0);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost); //grab data from GPU and
put it back into cpu memory

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
    sumTime += time;
    //printf("cudaMemcpy Device to Host elapsed time:  %3.3f ms \n", time);
```

```
    overheadTime = sumTime - gpuTime; //find out how much of the time was not
calculation time

    printf("GPU elapsed time:  %3.3f ms (%3.3f ms was overhead, %3.3f ms was
calculation)\n", sumTime, overheadTime, gpuTime);

    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C); //free the data
}


int main(int  argc, char* argv[])
{

    int numberArray[] ={32, 212, 8192, 81408, 81049};
    int counterValue;
    if(argc == 0)
    {
        counterValue = 5;
    }
    else
    {
        counterValue = 1;
        numberArray[0]= atoi(argv[1]);
        //printf("Entered Value:%d aka %s\n ",numberArray[0],argv[1]);
    }

    int n = 0;
    for (int counter = 0; counter < counterValue; counter ++)
    {
        cudaDeviceReset();
        n = numberArray[counter];
        int *a, *b, *c, *d;
        int size = n * sizeof(int);
        a = (int *)malloc(size);
        b = (int *)malloc(size);
        c = (int *)malloc(size);

        for (int i = 0; i < n; ++i) //for this test it simples adds two of the
same number
        {
            a[i] = i;
            b[i] = i;
            c[i] = 0;
        }
```

```
        printf("n = %d\n", n);

        //CPUtest(a, b, c, n);

        GPUtest(a, b, c, n);


        free(a);
        free(b);
        free(c);
    }
}
```

```
#include <stdio.h>
#include <stdlib.h>

void cpuVectorAdd(int *a, int *b, int *c, int n)
{
    int i;

    for (i = 0; i < n; i++)
    {
        c[i] = a[i] + b[i];
    }
}

__global__ void gpuVectorAdd(int *a, int *b, int *c, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n)
    {
        c[i] = a[i] + b[i];
    }
}


void CPUtest(int *a, int *b, int*c, int n)
{
    float time;
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start, 0);

    cpuVectorAdd(a, b, c, n);
    cudaDeviceSynchronize();//simply there to balance the test

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
    printf("CPU elapsed time:  %3.3f ms \n", time);
}


void GPUtest(int *h_A, int *h_B, int*h_C, int n)
{
```

simpleCUFFT.cu

```
/* Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *  * Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *  * Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *  * Neither the name of NVIDIA CORPORATION nor the names of its
 *    contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ``AS IS'' AND ANY
 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
 * OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

/* Example showing the use of CUFFT for fast 1D-convolution using FFT. */

// includes, system
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//#include <fftw3.h>

// includes, project
#include <cuda_runtime.h>
#include <cufft.h>
#include <cufftXt.h>
#include <helper_cuda.h>
#include <helper_functions.h>
```

```
// Complex data type
typedef float2 Complex;
static __device__ __host__ inline Complex ComplexAdd(Complex, Complex);
static __device__ __host__ inline Complex ComplexScale(Complex, float);
static __device__ __host__ inline Complex ComplexMul(Complex, Complex);
static __global__ void ComplexPointwiseMulAndScale(Complex *, const Complex *,
                                                   int, float);

// Filtering functions
void Convolve(const Complex *, int, const Complex *, int, Complex *);

// Padding functions
int PadData(const Complex *, Complex **, int, const Complex *, Complex **, int);

////////////////////////////////////////////////////////////////////////////////
// declaration, forward
float runTest(int argc, char **argv, int SIGNAL_SIZE);


#define REAL 0
#define IMAG 1
// The filter size is assumed to be a number smaller than the signal size
#define FILTER_KERNEL_SIZE 11

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////

/*
void acquire_from_somewhere(fftw_complex* signal, int NUM_POINTS) {


  int i;
  for (i = 0; i < NUM_POINTS; ++i) {
    double theta = (double)i / (double)NUM_POINTS * M_PI;

    signal[i][REAL] = 1.0 * cos(10.0 * theta) +
      0.5 * cos(25.0 * theta);

    signal[i][IMAG] = 1.0 * sin(10.0 * theta) +
      0.5 * sin(25.0 * theta);
  }
}
```

```
float cpuTest(int NUM_POINTS) {
  fftw_complex signal[NUM_POINTS];
  fftw_complex result[NUM_POINTS];

  fftw_plan plan = fftw_plan_dft_1d(NUM_POINTS,
    signal,
    result,
    FFTW_FORWARD,
    FFTW_ESTIMATE);

  acquire_from_somewhere(signal, NUM_POINTS);

  float cputime;
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  cudaEventRecord(start, 0);

  fftw_execute(plan);

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&cputime, start, stop);

  fftw_destroy_plan(plan);

  return cputime;
}
*/

int main(int argc, char **argv)
{
  FILE *f = fopen("FFT.txt", "a");
  if (f == NULL)
  {
    printf("Error opening file!\n");
    exit(1);
  }

  int n = 10;

  while (n < 1000000)
```

```
  {

    cudaDeviceReset();
    fprintf(f, "GPU(%d)\n", n);

    float time, exetime, cputime;
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start, 0);

    exetime = runTest(argc, argv, n);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
    printf("GPU total time:  %3.3f ms \n", time);

    fprintf(f, "%3.3f, %3.3f \n", exetime, time);

    /*fprintf(f, "CPU(%d)\n", n);

    cputime = cpuTest(n);

    fprintf(f, "%3.3f\n", cputime);*/

    n *= 10;
  }
  fclose(f);

}

////////////////////////////////////////////////////////////////////////////////
//! Run a simple test for CUDA
////////////////////////////////////////////////////////////////////////////////
float runTest(int argc, char **argv, int SIGNAL_SIZE) {

  printf("[simpleCUFFT] is starting...\n");

  float time;
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
```

```
findCudaDevice(argc, (const char **)argv);

// Allocate host memory for the signal
Complex *h_signal =
    reinterpret_cast<Complex *>(malloc(sizeof(Complex) * SIGNAL_SIZE));

// Initialize the memory for the signal
for (unsigned int i = 0; i < SIGNAL_SIZE; ++i) {
  h_signal[i].x = rand() / static_cast<float>(RAND_MAX);
  h_signal[i].y = 0;
}



// Allocate host memory for the filter
Complex *h_filter_kernel =
    reinterpret_cast<Complex *>(malloc(sizeof(Complex) * FILTER_KERNEL_SIZE));

// Initialize the memory for the filter
for (unsigned int i = 0; i < FILTER_KERNEL_SIZE; ++i) {
  h_filter_kernel[i].x = rand() / static_cast<float>(RAND_MAX);
  h_filter_kernel[i].y = 0;
}

// Pad signal and filter kernel
Complex *h_padded_signal;
Complex *h_padded_filter_kernel;
int new_size =
    PadData(h_signal, &h_padded_signal, SIGNAL_SIZE, h_filter_kernel,
            &h_padded_filter_kernel, FILTER_KERNEL_SIZE);
int mem_size = sizeof(Complex) * new_size;

// Allocate device memory for signal
Complex *d_signal;
checkCudaErrors(cudaMalloc(reinterpret_cast<void **>(&d_signal), mem_size));
// Copy host memory to device
checkCudaErrors(
    cudaMemcpy(d_signal, h_padded_signal, mem_size, cudaMemcpyHostToDevice));

// Allocate device memory for filter kernel
Complex *d_filter_kernel;
checkCudaErrors(
    cudaMalloc(reinterpret_cast<void **>(&d_filter_kernel), mem_size));

// Copy host memory to device
```

```
  checkCudaErrors(cudaMemcpy(d_filter_kernel, h_padded_filter_kernel, mem_size,
                             cudaMemcpyHostToDevice));

  // CUFFT plan simple API
  cufftHandle plan;
  checkCudaErrors(cufftPlan1d(&plan, new_size, CUFFT_C2C, 1));

  // CUFFT plan advanced API
  cufftHandle plan_adv;
  size_t workSize;
  long long int new_size_long = new_size;

  checkCudaErrors(cufftCreate(&plan_adv));
  checkCudaErrors(cufftXtMakePlanMany(plan_adv, 1, &new_size_long, NULL, 1, 1,
                                      CUDA_C_32F, NULL, 1, 1, CUDA_C_32F, 1,
                                      &workSize, CUDA_C_32F));
  printf("Temporary buffer size %li bytes\n", workSize);

  // Transform signal and kernel
  printf("Transforming signal cufftExecC2C\n");
  checkCudaErrors(cufftExecC2C(plan, reinterpret_cast<cufftComplex *>(d_signal),
                               reinterpret_cast<cufftComplex *>(d_signal),
                               CUFFT_FORWARD));
  checkCudaErrors(cufftExecC2C(
      plan_adv, reinterpret_cast<cufftComplex *>(d_filter_kernel),
      reinterpret_cast<cufftComplex *>(d_filter_kernel), CUFFT_FORWARD));

//start timer
  cudaEventRecord(start, 0);

  // Multiply the coefficients together and normalize the result
  printf("Launching ComplexPointwiseMulAndScale<<< >>>\n");
  ComplexPointwiseMulAndScale<<<32, 256>>>(d_signal, d_filter_kernel, new_size,
                                           1.0f / new_size);

  cudaDeviceSynchronize();

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&time, start, stop);
  printf("GPU execution elapsed time:  %3.3f ms \n", time);

  // Check if kernel execution generated and error
  getLastCudaError("Kernel execution failed [ ComplexPointwiseMulAndScale ]");
```

```
  // Transform signal back
  printf("Transforming signal back cufftExecC2C\n");
  checkCudaErrors(cufftExecC2C(plan, reinterpret_cast<cufftComplex *>(d_signal),
                               reinterpret_cast<cufftComplex *>(d_signal),
                               CUFFT_INVERSE));

  // Copy device memory to host
  Complex *h_convolved_signal = h_padded_signal;
  checkCudaErrors(cudaMemcpy(h_convolved_signal, d_signal, mem_size,
                             cudaMemcpyDeviceToHost));

  // Allocate host memory for the convolution result
  Complex *h_convolved_signal_ref =
      reinterpret_cast<Complex *>(malloc(sizeof(Complex) * SIGNAL_SIZE));

  // Convolve on the host
  Convolve(h_signal, SIGNAL_SIZE, h_filter_kernel, FILTER_KERNEL_SIZE,
           h_convolved_signal_ref);

  // check result
  bool bTestResult = sdkCompareL2fe(
      reinterpret_cast<float *>(h_convolved_signal_ref),
      reinterpret_cast<float *>(h_convolved_signal), 2 * SIGNAL_SIZE, 1e-5f);

  // Destroy CUFFT context
  checkCudaErrors(cufftDestroy(plan));
  checkCudaErrors(cufftDestroy(plan_adv));

  // cleanup memory
  free(h_signal);
  free(h_filter_kernel);
  free(h_padded_signal);
  free(h_padded_filter_kernel);
  free(h_convolved_signal_ref);
  checkCudaErrors(cudaFree(d_signal));
  checkCudaErrors(cudaFree(d_filter_kernel));

  return time;
  //exit(bTestResult ? EXIT_SUCCESS : EXIT_FAILURE);
}

// Pad data
int PadData(const Complex *signal, Complex **padded_signal, int signal_size,
            const Complex *filter_kernel, Complex **padded_filter_kernel,
            int filter_kernel_size) {
```

```
  int minRadius = filter_kernel_size / 2;
  int maxRadius = filter_kernel_size - minRadius;
  int new_size = signal_size + maxRadius;

  // Pad signal
  Complex *new_data =
      reinterpret_cast<Complex *>(malloc(sizeof(Complex) * new_size));
  memcpy(new_data + 0, signal, signal_size * sizeof(Complex));
  memset(new_data + signal_size, 0, (new_size - signal_size) * sizeof(Complex));
  *padded_signal = new_data;

  // Pad filter
  new_data = reinterpret_cast<Complex *>(malloc(sizeof(Complex) * new_size));
  memcpy(new_data + 0, filter_kernel + minRadius, maxRadius * sizeof(Complex));
  memset(new_data + maxRadius, 0,
         (new_size - filter_kernel_size) * sizeof(Complex));
  memcpy(new_data + new_size - minRadius, filter_kernel,
         minRadius * sizeof(Complex));
  *padded_filter_kernel = new_data;

  return new_size;
}

////////////////////////////////////////////////////////////////////////////////
// Filtering operations
////////////////////////////////////////////////////////////////////////////////

// Computes convolution on the host
void Convolve(const Complex *signal, int signal_size,
              const Complex *filter_kernel, int filter_kernel_size,
              Complex *filtered_signal) {
  int minRadius = filter_kernel_size / 2;
  int maxRadius = filter_kernel_size - minRadius;

  // Loop over output element indices
  for (int i = 0; i < signal_size; ++i) {
    filtered_signal[i].x = filtered_signal[i].y = 0;

    // Loop over convolution indices
    for (int j = -maxRadius + 1; j <= minRadius; ++j) {
      int k = i + j;

      if (k >= 0 && k < signal_size) {
        filtered_signal[i] =
            ComplexAdd(filtered_signal[i],
```

```
                        ComplexMul(signal[k], filter_kernel[minRadius - j]));
      }
    }
  }
}

////////////////////////////////////////////////////////////////////////////////
// Complex operations
////////////////////////////////////////////////////////////////////////////////

// Complex addition
static __device__ __host__ inline Complex ComplexAdd(Complex a, Complex b) {
  Complex c;
  c.x = a.x + b.x;
  c.y = a.y + b.y;
  return c;
}

// Complex scale
static __device__ __host__ inline Complex ComplexScale(Complex a, float s) {
  Complex c;
  c.x = s * a.x;
  c.y = s * a.y;
  return c;
}

// Complex multiplication
static __device__ __host__ inline Complex ComplexMul(Complex a, Complex b) {
  Complex c;
  c.x = a.x * b.x - a.y * b.y;
  c.y = a.x * b.y + a.y * b.x;
  return c;
}

// Complex pointwise multiplication
static __global__ void ComplexPointwiseMulAndScale(Complex *a, const Complex *b,
                                                   int size, float scale) {
  const int numThreads = blockDim.x * gridDim.x;
  const int threadID = blockIdx.x * blockDim.x + threadIdx.x;

  for (int i = threadID; i < size; i += numThreads) {
    a[i] = ComplexScale(ComplexMul(a[i], b[i]), scale);
  }
}
```

fftw.c

```c
/* Start reading here */

#include <fftw3.h>
#include <sys/time.h>
#include <stdlib.h>
//#define NUM_POINTS 64


/* Never mind this bit */

#include <stdio.h>
#include <math.h>

#define REAL 0
#define IMAG 1

void acquire_from_somewhere(fftw_complex* signal, int NUM_POINTS) {
    /* Generate two sine waves of different frequencies and
     * amplitudes.
     */

    int i;
    for (i = 0; i < NUM_POINTS; ++i) {
        double theta = (double)i / (double)NUM_POINTS * M_PI;

        signal[i][REAL] = 1.0 * cos(10.0 * theta) +
                          0.5 * cos(25.0 * theta);

        signal[i][IMAG] = 1.0 * sin(10.0 * theta) +
                          0.5 * sin(25.0 * theta);
    }
}


/* Resume reading here */

float cpuTest(int NUM_POINTS) {
    fftw_complex signal[NUM_POINTS];
    fftw_complex result[NUM_POINTS];

    fftw_plan plan = fftw_plan_dft_1d(NUM_POINTS,
                                      signal,
                                      result,
```

```
                                    FFTW_FORWARD,
                                    FFTW_ESTIMATE);

    acquire_from_somewhere(signal, NUM_POINTS);

    struct timeval t1, t2;
    double elapsedTime;

    // start timer
    gettimeofday(&t1, NULL);

    fftw_execute(plan);

    // stop timer
    gettimeofday(&t2, NULL);

    // compute and print the elapsed time in millisec
    elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000.0;      // sec to ms
    elapsedTime += (t2.tv_usec - t1.tv_usec) / 1000.0;   // us to ms

    fftw_destroy_plan(plan);

    return elapsedTime;
}

int main()
{
    FILE *f = fopen("FFT.txt", "a");
    if (f == NULL)
    {
        printf("Error opening file!\n");
        exit(1);
    }

    int n = 10;

    while (n < 1000000)
    {
        float cputime;

        fprintf(f, "CPU(%d)\n", n);

        cputime = cpuTest(n);

        fprintf(f, "%3.3f\n", cputime);
```

```
        n *= 10;
    }
    fclose(f);
}
```

Smiscript.sh

```
#!/bin/bash

echo "running Nvidia-smi"

while true; do
nvidia-smi | grep  MiB
done
```

memorychecker.sh

```
#!/bin/bash

trap "kill 0" EXIT

(exec "./smiscript.sh" &)
echo running simpleTest 10
./simpleTest 10
echo running simpleTest 100
./simpleTest 100
echo running simpleTest 1000
./simpleTest 1000
echo running simpleTest 10000
./simpleTest 10000
echo running simpleTest 100000
./simpleTest 100000
echo running simpleTest 1000000
./simpleTest 1000000
echo running simpleTest 10000000
./simpleTest 10000000
echo tests complete
wait
```

runMemorychecker.sh

```
#!/bin/bash

rm memoryCheckerOutput.txt
(exec "./memoryChecker.sh" > memoryCheckerOutput.txt)
```